Easy, Effective, Efficient: GPU Programming in Python with PyOpenCL and PyCUDA

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

PASI: The Challenge of Massive Parallelism
Lecture 2 · January 5, 2011
import pyopencl as cl, numpy

a = numpy.random.rand(256**3).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_dev, a)

prg = cl.Program(ctx, ""
    __kernel void twice(__global float *a) {
        a[get_global_id(0)] *= 2;
    }
"").build()

prg.twice(queue, a.shape, (1,), a_dev)
CL consists of two parts:

- **Host-side “runtime”:**
  
  C: `#include "CL/cl.h"`, `libOpenCL.so`
  
  Python:
  
  `import pyopencl as cl`

- **Device-side programming language**
  
  A dialect of C99 (also when using PyOpenCL)

Today: A better look at both of those, with an emphasis on PyOpenCL
Outline

1. The OpenCL Runtime
2. Device Language
3. OpenCL implementations
Outline

1 The OpenCL Runtime
2 Device Language
3 OpenCL implementations
Figure 2.1 - OpenCL UML Class Diagram

Credit: Khronos Group
"Platform": a collection of devices, all from the same vendor.

All devices in a platform use the same CL driver/implementation.

Multiple platforms can be used from one program → ICD.

libOpenCL.so: ICD loader

/etc/OpenCL/vendors/somename.icd: Plain text file with name of .so containing CL implementation.
CL “Compute Device”

CL Compute Devices:
- CPUs, GPUs, accelerators, . . .
  - Anything that fits the programming model.
- A processor die with an interface to off-chip memory
- Can get list of devices from platform.
context = cl.Context(devices=None | [dev1, dev2], dev_type=None)
context = cl.create_some_context( interactive =True)

- Spans one or more Devices
- Create from device type or list of devices
  - See docs for cl.Platform, cl.Device
- dev_type: DEFAULT, ALL, CPU, GPU
- Needed to...
  - ...allocate Memory Objects
  - ...create and build Programs
  - ...host Command Queues
  - ...execute Grids
OpenCL: Command Queues

- Host and Device run asynchronously
- Host submits to queue:
  - Computations
  - Memory Transfers
  - Sync primitives
  - ...  
- Host can wait for drained queue
- Profiling
Command Queues and Events

```python
queue = cl.CommandQueue(context, device=None,
                         properties =None | [(prop, value ),...])
```

- Attached to single device
- `cl.command_queue.properties` ...
  - `OUT_OF_ORDER_EXEC_MODE_ENABLE`:
    - Do not force sequential execution
  - `PROFILING_ENABLE`:
    - Gather timing info
import pyopencl as cl

platforms = cl.get_platforms()
my_platform = platforms[0]
print my_platform.vendor

devices = my_platform.get_devices()
my_device = devices[0]
print my_device.name

ctx = cl.Context([my_device])

cpq = cl.command_queue_properties
queue = cl.CommandQueue(ctx, my_device, cpq.PROFILING_ENABLE)

Simple version:

ctx2 = cl.create_some_context()
queue2 = cl.CommandQueue(ctx)
Command Queues and Events

```
event = cl.enqueue_XXX(queue, ..., wait_for=[evt1, evt2])
```

Every enqueue operation returns an *Event*.

Also possible: Operation-less events ("Markers")

- Wait (evt.wait()), polling
- Specify dependencies
  - Every enqueue operation takes a list arg wait_for of dependencies.
- Profile
  - event.profile....
    - QUEUED, SUBMIT
    - START, END
  (time stamp in ns)
Profiling example

```python
start_event = cl.enqueue_marker(queue)

# enqueue some commands

stop_event = cl.enqueue_marker(queue)
stop_event.wait()

elapsed_seconds = 1e-9*(
    start_event.profile.END - start_event.profile.END)

# --- OR ---

op_event = knl(queue, global_size, grp_size, args ...)
op_event.wait()
elapsed_seconds = 1e-9*(
    op_event.profile.END - start_event.profile.START)
```
Capturing Dependencies

\[
B = f(A) \\
C = g(B) \\
E = f(C) \\
F = h(C) \\
G = g(E,F) \\
P = p(B) \\
Q = q(B) \\
R = r(G,P,Q)
\]
Capturing Dependencies

- Switch queue to out-of-order mode!
- Specify as list of events using `wait_for=` optional keyword to `enqueue_XXX`.
- Can also enqueue barrier.
- Common use case: Transmit/receive from other MPI ranks.
- Possible on Nv Fermi: Submit parallel work to increase machine use.
Memory Objects: Buffers

buf = cl.Buffer(context, flags, size=0, hostbuf=None)

- Chunk of device memory
- No type information: “Bag of bytes”
- Observe: Not tied to device.
  → no fixed memory address
  → pointers do not survive kernel launches
  → movable between devices
- flags:
  - READ_ONLY/WRITABLE/READ_WRITE
  - {ALLOC,COPY,USE}_HOST_PTR
Memory Objects: Buffers

```python
buf = cl.Buffer(context, flags, size=0, hostbuf=None)
```

**COPY_HOST_PTR:**
- Use `hostbuf` as initial content of buffer

**USE_HOST_PTR:**
- `hostbuf` is the buffer.
- Caching in device memory is allowed.

**ALLOC_HOST_PTR:**
- New host memory (unrelated to `hostbuf`) is visible from device and host.
Memory Objects: Buffers

```python
buf = cl.Buffer(context, flags, size=0, hostbuf=None)
```

- Specify `hostbuf` or `size` (or both)
- `hostbuf`: Needs Python Buffer Interface
e.g. `numpy.ndarray`, `str`
  - Important: Memory layout matters
- Passed to device code as pointers
  (e.g. `float *`, `int *`)
- `enqueue_{read,write}_buffer`
  `queue, buf, hostbuf`
- Can be mapped into host address space:
  `cl.MemoryMap`
Command Queues and Buffers: A Crashy Puzzle

✅ OK

```python
a = numpy.random.rand(256**3).astype(numpy.float32)
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_dev, a, is_blocking=False)
```
Command Queues and Buffers: A Crashy Puzzle

✔ OK

```python
a = numpy.random.rand(256**3).astype(numpy.float32)
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_dev, a, is_blocking=False)
```

✗ Crash

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_write_buffer(queue, a_dev,
    numpy.random.rand(256**3).astype(numpy.float32),
    is_blocking=False)
```
Command Queues and Buffers: A Crashy Puzzle

✅ OK

```python
a = numpy.random.rand(256**3).astype(numpy.float32)
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_dev, a,
    is_blocking =False)
```

❌ Crash

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_write_buffer(queue, a_dev,
    numpy.random.rand(256**3).astype(numpy.float32),
    is_blocking =False)
```

✅ OK

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_write_buffer(queue, a_dev,
    numpy.random.rand(256**3).astype(numpy.float32),
    is_blocking =True)
```
Command Queues and Buffers: A Crashy Puzzle

✓ OK (usually!)

\[
a = \text{numpy.random.rand}(256**3).astype(\text{numpy.float32})
\]
\[
a_{\text{dev}} = \text{cl.Buffer}(\text{ctx}, \text{cl.mem_flags.READ\_WRITE}, \text{size}=a.\text{nbytes})
\]
\[
\text{cl.enqueue_write_buffer}(\text{queue}, a_{\text{dev}}, a, \text{is\_blocking} = \text{False})
\]

✗ Crash

\[
a_{\text{dev}} = \text{cl.Buffer}(\text{ctx}, \text{cl.mem_flags.READ\_WRITE}, \text{size}=256**3*4)
\]
\[
\text{cl.enqueue_write_buffer}(\text{queue}, a_{\text{dev}}, \text{numpy.random.rand}(256**3).astype(\text{numpy.float32}), \text{is\_blocking} = \text{False})
\]

✓ OK

\[
a_{\text{dev}} = \text{cl.Buffer}(\text{ctx}, \text{cl.mem_flags.READ\_WRITE}, \text{size}=256**3*4)
\]
\[
\text{cl.enqueue_write_buffer}(\text{queue}, a_{\text{dev}}, \text{numpy.random.rand}(256**3).astype(\text{numpy.float32}), \text{is\_blocking} = \text{True})
\]
prg = cl.Program(context, src)

- **src**: OpenCL device code
  - Derivative of C99
  - Functions with `__kernel` attribute can be invoked from host
- `prg.build(options="", devices=None)`
- `kernel = prg.kernel_name`
- `kernel(queue, (Gx, Gy, Gz), (Lx, Ly, Lz), arg, ..., wait_for=None)`
kernel(queue, (Gx,Gy,Gz), (Sx,Sy,Sz), arg, ..., wait_for=None)

arg may be:

- None (a NULL pointer)
- numpy sized scalars:
  numpy.int64, numpy.float32, ...
- Anything with buffer interface:
  numpy.ndarray, str
- Buffer Objects
- Also: cl.Image, cl.Sampler, cl.LocalMemory
Program Objects

kernel (queue, (Gx,Gy,Gz), (Sx,Sy,Sz), arg, ..., wait_for = None)

Explicitly sized scalars:

✖ Annoying, error-prone.

Better:

kernel.set_scalar_arg_dtypes([numpy.int32, None, numpy.float32])

Use None for non-scalars.
OpenCL Object Diagram

Credit: Khronos Group

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Outline

1. The OpenCL Runtime
2. Device Language
   - Synchronization
   - Extensions
3. OpenCL implementations
OpenCL device language is C99, with these differences:

- Index getters
- Memory space qualifiers
- Vector data types
- Many generic (‘overloaded’) math functions including fast native... varieties.
- Synchronization
- Recursion
- `malloc()`
### OpenCL ↔ CUDA: A dictionary

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>Grid</td>
</tr>
<tr>
<td>Work Group</td>
<td>Block</td>
</tr>
<tr>
<td>Work Item</td>
<td>Thread</td>
</tr>
<tr>
<td>__kernel</td>
<td><strong>global</strong></td>
</tr>
<tr>
<td>__global</td>
<td><strong>device</strong></td>
</tr>
<tr>
<td>__local</td>
<td><strong>shared</strong></td>
</tr>
<tr>
<td>__private</td>
<td><strong>local</strong></td>
</tr>
<tr>
<td>imagedd_t</td>
<td>texture&lt;type, n, ...&gt;</td>
</tr>
<tr>
<td>barrier (LMF)</td>
<td>__syncthreads()</td>
</tr>
<tr>
<td>get_local_id(012)</td>
<td>threadIdx.xyz</td>
</tr>
<tr>
<td>get_group_id(012)</td>
<td>blockIdx.xyz</td>
</tr>
<tr>
<td>get_global_id(012)</td>
<td>– (reimplement)</td>
</tr>
</tbody>
</table>
## Address Space Qualifiers

<table>
<thead>
<tr>
<th>Type</th>
<th>Per</th>
<th>Access</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>work item</td>
<td>R/W</td>
<td>1 or 1000</td>
</tr>
<tr>
<td>local</td>
<td>group</td>
<td>R/W</td>
<td>2</td>
</tr>
<tr>
<td>global</td>
<td>grid</td>
<td>R/W</td>
<td>1000</td>
</tr>
<tr>
<td>constant</td>
<td>grid</td>
<td>R/O</td>
<td>1-1000</td>
</tr>
<tr>
<td>image&lt;nd_t</td>
<td>grid</td>
<td>R(/W)</td>
<td>1000</td>
</tr>
</tbody>
</table>

Important: Don’t “choose one” type of memory. Successful algorithms combine many types’ strengths.

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
### Address Space Qualifiers

<table>
<thead>
<tr>
<th>Type</th>
<th>Per</th>
<th>Access</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>work item</td>
<td>R/W</td>
<td>1 or 1000</td>
</tr>
<tr>
<td>local</td>
<td>group</td>
<td>R/W</td>
<td>2</td>
</tr>
<tr>
<td>global</td>
<td>grid</td>
<td>R/W</td>
<td>1000</td>
</tr>
<tr>
<td>constant</td>
<td>grid</td>
<td>R/O</td>
<td>1-1000</td>
</tr>
<tr>
<td><code>image&lt;nd_t</code></td>
<td>grid</td>
<td>R(/W)</td>
<td>1000</td>
</tr>
</tbody>
</table>
## Address Space Qualifiers

<table>
<thead>
<tr>
<th>Type</th>
<th>Per</th>
<th>Access</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>work item</td>
<td>R/W</td>
<td>1 or 1000</td>
</tr>
<tr>
<td>local</td>
<td>group</td>
<td>R/W</td>
<td>2</td>
</tr>
<tr>
<td>global</td>
<td>grid</td>
<td>R/W</td>
<td>1000 Not cached</td>
</tr>
<tr>
<td>constant</td>
<td>grid</td>
<td>R/O</td>
<td>1-1000 Cached</td>
</tr>
<tr>
<td>image(nd_t)</td>
<td>grid</td>
<td>R(/W)</td>
<td>1000 Spatially cached</td>
</tr>
</tbody>
</table>

### Important

Don’t “choose one” type of memory. Successful algorithms combine many types’ strengths.
CL vector data types

float \textit{n} vec \textit{(n=1,2,3,4,8,16)} (also for double and integer types) Components:

- \textit{vec.s012...abcdef} (or \textit{xyzw})
- \textit{vec.s3120} (Swizzling)
- \textit{vec.s024} = (float3)(1,2,3);
  (Lvalue, Literals)

Usage:

- Elementwise operations (+, -, \textit{sin} (generic!),...)
- float\textit{n vloadn/vstoren}\textit{(offset, float *)} (aligned!)
- dot/distance

Using CPU implementation: One of the sanest ways of using SSE/vector intrinsics!
Outline

1. The OpenCL Runtime

2. Device Language
   - Synchronization
   - Extensions

3. OpenCL implementations
Recap: Concurrency and Synchronization

GPUs have layers of concurrency.
Each layer has its synchronization primitives.
Recap: Concurrency and Synchronization

GPUs have layers of concurrency.
Each layer has its synchronization primitives.

- **Intra-group:**
  - `barrier(...)`,
  - `mem_fence(...)`

  ```
  ... = CLK_{LOCAL,GLOBAL}_MEM_FENCE
  ```

- **Inter-group:**
  - Kernel launch

- **CPU-GPU:**
  - Command queues, Events
Golden Rule:

Results of the algorithm must be independent of the order in which work groups are executed.
Synchronization between Groups

**Golden Rule:**

Results of the algorithm must be independent of the order in which work groups are executed.

**Consequences:**

- Work groups may read the same information from global memory.
- But: Two work groups may not validly write different things to the same global memory.
- Kernel launch serves as
  - Global barrier
  - Global memory fence
What is a Barrier?
Synchronizaton

What is a Barrier?

---
What is a Barrier?
Synchronisation

What is a Barrier?
Synchronisation

What is a Barrier?
Synchronization

What is a Barrier?
Synchronisation

What is a Barrier?
Synchronization

What is a Memory Fence?
Synchronization

What is a Memory Fence?

write 18

17
Synchronization

What is a Memory Fence?
Synchronization

What is a Memory Fence?

write 18

17

read 17
Synchronization

What is a Memory Fence?

write 18

17
Synchronization

What is a Memory Fence?

write 18

18
Synchronization

What is a Memory Fence?
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Synchronization

What is a Memory Fence? An ordering restriction for memory access.

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible! → Interruptible!

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected

Protected

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected

How?

atomic_{add,inc,cmpxchg,\ldots}(int *global, \text{int value});
Outline

1. The OpenCL Runtime

2. Device Language
   - Synchronization
   - Extensions

3. OpenCL implementations
Extensions: Future-proof CL

Similar extensions mechanism to OpenGL.

Two mechanisms:

- Runtime:
  - `cl_ext.h` header
  - availability checkable via `#ifdef device.extensions`

- Device language:
  - `#pragma OPENCL EXTENSION name : enable`

Important extension:

- `cl_khr_fp64`

Vendor and ‘official’ extensions.
Outline

1. The OpenCL Runtime
2. Device Language
3. OpenCL implementations
The Nvidia CL implementation

Targets only GPUs

Notes:

- Nearly identical to CUDA
  - No native C-level JIT in CUDA (→ PyCUDA)
- Page-locked memory:
  Use CL_MEM_ALLOC_HOST_PTR.
  (Careful: double meaning)
- No linear memory texturing
- CUDA device emulation mode deprecated
  → Use AMD CPU CL (faster, too!)
The Apple CL implementation

Targets CPUs and GPUs

General notes:

- Different header name
  `OpenCL/cl.h` instead of `CL/cl.h`
  Use `-framework OpenCL` for C access.

- Beware of imperfect compiler cache implementation
  (ignores include files)

CPU notes:

- One work item per processor

GPU similar to hardware vendor implementation.
(New: Intel w/ Sandy Bridge)
The AMD CL implementation

Targets CPUs and GPUs (from both AMD and Nvidia)

GPU notes:
- Wide SIMD groups (64)
- Native 4/5-wide vectors
  - But: very flop-heavy machine, may ignore vectors for memory-bound workloads
- $\rightarrow$ Both implicit and explicit SIMD

CPU notes:
- Many work items per processor (emulated)
- cl_amd_printf
Questions?
Image Credits

- Context: sxc.hu/svilen001
- Queue: sxc.hu/cobrasoft
- Check mark: sxc.hu/bredmaker
- RAM stick: sxc.hu/gobran11
- CPU: sxc.hu/dimshik
- Dictionary: sxc.hu/topfer
- Dominoes: sxc.hu/rolve
- Onions: flickr.com/darwinbell
- Bricks: sxc.hu/guitargoa
- Nvidia logo: Nvidia Corporation
- Apple logo: Apple Corporation
- AMD logo: AMD Corporation